(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 660 231 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **94119549.7**

(51) Int. Cl.[6]: **G06F 9/44**

(22) Date of filing: **09.12.94**

(30) Priority: **13.12.93 US 166976**

(43) Date of publication of application:
28.06.95 Bulletin 95/26

(84) Designated Contracting States:
DE FR GB

(71) Applicant: **MICROSOFT CORPORATION**
One Microsoft Way
Redmond,
Washington 98052-6399 (US)

(72) Inventor: **Stutz, David S.**
19610 N.E. 116th,
Redmond
Washington 98053 (US)
Inventor: **Zimmerman, Christopher A.**
3006 West Lake Sammamish Parkway S.E.
Bellevue,
Washington 98008 (US)

(74) Representative: **Patentanwälte Grünecker,**
Kinkeldey, Stockmair & Partner
Maximilianstrasse 58
D-80538 München (DE)

(54) Method and system for dynamically generating object connections.

(57) A method and system for dynamically generating object connections is provided. In a preferred embodiment, a connection can be generated between a source object and a sink object using a connection point object. A source object has connection point objects where each connection point object corresponds to a particular interface. A sink object implements one or more notification interfaces for connecting to a source object. A connection point object of a source object can connect to multiple notification interfaces, which belong to one or more sink objects. A connection point object keeps track of pointers to the notification interfaces to which it has been connected. In order to generate a connection, a sink object requests from a source object a connection point object corresponding to a particular interface. The source object determines whether it supports such a connection point object, and if so returns a pointer to the connection point interface of the determined connection point object. The sink object then requests to be connected to the connection point object using the returned connection point interface pointer and passes a reference to a notification interface of the sink object corresponding to the particular interface. The connection point object then stores the reference to the notification interface of the sink object, creating a connection between the sink object and the source object. At some later time, the source object can utilize the connection to notify the sink object through the connected notification interfaces.

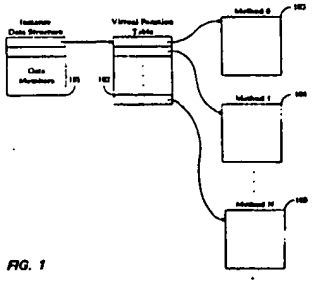

FIG. 1

EP 0 660 231 A2

base class CIRCLE.

```
class CIRCLE_FILL : CIRCLE
{ public:
        int pattern;
        void fill();
};
```

This declaration specifies that class CIRCLE_FILL includes all the data and function members that are in class CIRCLE in addition to those data and function members introduced in the declaration of class CIRCLE_FILL, that is, data member pattern and function member fill. In this example, class CIRCLE_FILL has data members x, y, radius, and pattern and function members draw and fill. Class CIRCLE_FILL is said to "inherit" the characteristics of class CIRCLE. A class that inherits the characteristics of another class is a derived class (e.g., CIRCLE_FILL). A class that does not inherit the characteristics of another class is a primary (root) class (e.g., CIRCLE). A class whose characteristics are inherited by another class is a base class (e.g., CIRCLE is a base class of CIRCLE_FILL). A derived class may inherit the characteristics of several classes, that is, a derived class may have several base classes. This is referred to as multiple inheritance.

A derived class may specify that a base class is to be inherited virtually. Virtual inheritance of a base class means that only one instance of the virtual base class exists in the derived class. For example, the following is an example of a derived class with two nonvirtual base classes.

```
class CIRCLE_1 : CIRCLE {...};
class CIRCLE_2 : CIRCLE {...};
class PATTERN : CIRCLE_1, CIRCLE_2{...};
```

In this declaration class PATTERN inherits class CIRCLE twice nonvirtually through classes CIRCLE_1 and CIRCLE_2. There are two instances of class CIRCLE in class PATTERN.

The following is an example of a derived class with two virtual base classes.

```
class CIRCLE_1 : virtual CIRCLE {...};
class CIRCLE_2 : virtual CIRCLE {...};
class PATTERN: CIRCLE_1, CIRCLE_2{...};
```

The derived class PATTERN inherits class CIRCLE twice virtually through classes CIRCLE_1 and CIRCLE_2. Since the class CIRCLE is virtually inherited twice, there is only one object of class CIRCLE in the derived class PATTERN. One skilled in the art would appreciate virtual inheritance can be very useful when the class derivation is more complex.

A class may also specify whether its function members are virtual. Declaring that a function member is virtual means that the function can be overridden by a function of the same name and type in a derived class. In the following example, the function draw is declared to be virtual in classes CIRCLE and CIRCLE_FILL.

compound document is a document that contains objects generated by various computer programs. (Typically, only the data members of the object and the class type are stored in a compound document.) For example, a word processing document that contains a spreadsheet object generated by a spreadsheet program is a compound document. A word processing program allows a user to embed a spreadsheet object (e.g., a cell) within a word processing document. To allow this embedding, the word processing program is compiled using the class definition of the object to be embedded to access function members of the embedded object. Thus, the word processing program would need to be compiled using the class definition of each class of objects that can be embedded in a word processing document. To embed an object of a new class into a word processing document, the word processing program would need to be recompiled with the new class definition. Thus, only objects of classes selected by the developer of the word processing program can be embedded. Furthermore, new classes can only be supported with a new release of the word processing program.

To allow objects of an arbitrary class to be embedded into compound documents, interfaces are defined through which an object can be accessed without the need for the word processing program to have access to the class definitions at compile time. An abstract class is a class in which there is at least one virtual function member with no implementation (a pure virtual function member). An interface is an abstract class with no data members and whose virtual functions are all pure. Thus, an interface provides a protocol for two programs to communicate. Interfaces are typically used for derivation: a program implements classes that provide implementations for the interfaces the classes are derived from. Thereafter, objects are created as instances of these derived classes.

The following class definition is an example definition of an interface. In this example, for simplicity of explanation, rather than allowing any class of object to be embedded in its documents, a word processing program allows spreadsheet objects to be embedded. Any spreadsheet object that provides this interface can be embedded, regardless of how the object is implemented. Moreover, any spreadsheet object, whether implemented before or after the word processing program is compiled, can be embedded.

```
class   ISpreadSheet
{       virtual void File() = 0;
        virtual void Edit() = 0;
        virtual void Formula() = 0;
        virtual void Format() = 0;
        virtual void GetCell (string RC, cell *pCell) = 0;
        virtual void Data() = 0;
}
```

The developer of a spreadsheet program would need to provide an implementation of the interface to allow the spreadsheet objects to be embedded in a word processing document.

When the word processing program embeds a spreadsheet object, the program needs access to the code that implements the interface for the spreadsheet object. To access the class code, each implementation is given a unique class identifier. For example, code implementing a spreadsheet object developed by Microsoft Corporation may have a class identifier of "MSSpreadsheet," while code implementing a spreadsheet object developed by another corporation may have a class identifier of "LTSSpreadsheet." A persistent registry in each computer system is maintained that maps each class identifier to the code that implements the class. Typically, when a spreadsheet program is installed on a computer system, the persistent registry is updated to reflect the availability of that class of spreadsheet objects. So long as a spreadsheet developer implements each function member defined by the interface and the persistent registry is maintained, the word processing program can embed instances of the developer's spreadsheet objects into a word processing document. The word processing program accesses the function members of the embedded spreadsheet objects without regard to who has implemented them or how they have been implemented.

Various spreadsheet developers may wish, however, to implement only certain function members. For example, a spreadsheet developer may not want to implement database support, but may want to support all other function members. To allow a spreadsheet developer to support only some of the function

Code Table 1

```
HRESULT XX::QueryInterface(REFIID iid, void **ppv)
{       ret = TRUE;
        switch (iid) {
          case IID_IBasic:
                *ppv = *pIBasic;
                break;
          case IID_IDatabase:
                *ppv = *pIDatabase;
                break;
          case IID_IUnknown:
                *ppv = this;
                break;
         default:
                ret = FALSE;
        }
        if (ret == TRUE) {AddRef();};
        return ret;
}
```

Code Table 1 contains pseudocode for C++ source code for a typical implementation of the method QueryInterface for class XX, which inherits the class IUnknown. If the spreadsheet object supports the IDatabase interface, then the method QueryInterface includes the appropriate case label within the switch statement. The variables pIBasic and pIDatabase point to a pointer to the virtual function tables of the IBasic and IDatabase interfaces, respectively. The method QueryInterface invokes to method AddRef (described below) to increment a reference count for the object of class XX when a pointer to an interface is returned.

Code Table 2

```
void XX::AddRef() {refcount++;}
void XX::Release() {if (--refcount==0) delete this;}
```

The interface IUnknown also defines the methods AddRef and Release, which are used to implement reference counting. Whenever a new reference to an interface is created, the method AddRef is invoked to increment a reference count of the object. Whenever a reference is no longer needed, the method Release is invoked to decrement the reference count of the object and, when the reference count goes to zero, to deallocate the object. Code Table 2 contains pseudocode for C++ source code for a typical implementation of the methods AddRef and Release for class XX, which inherits the class IUnknown.

The IDatabase interface and IBasic interface inherit the IUnknown interface. The following definitions illustrate the use of the IUnknown interface.

managed by the connection point object. Later, the source object determines what notification interfaces have been stored in a particular connection point object and invokes a particular method of each stored notification interface to notify each sink object that has connected a notification interface. Such notification typically occurs in response to an event, for example, movement from a user input device.

Brief Description of the Drawings

Figure 1 is a block diagram illustrating typical data structures used to represent an object.

Figure 2 is a symbolic representation of a spreadsheet object.

Figure 3 is a block diagram of a preferred connection mechanism architecture.

Figure 4 is a block diagram of a connection between a source object, a delegate object and a sink object.

Figure 5 is a block diagram of a visual programming environment display used to create an open file dialog box for an application program.

Figure 6 is a block diagram of object connections and data structures after connecting the objects shown in Figure 5 using the present invention.

Figure 7 is a flow diagram of a function SetUpConnection for connecting a specified sink object to a specified source object for a specified notification interface.

Figure 8 is a flow diagram for the method FindConnectionPoint of the IConnectionPointContainer interface.

Figure 9 is a flow diagram of a method that uses an established connection between a source object and a sink object.

Figure 10 is a flow diagram of a function defined by a sink object to disconnect a specified notification interface.

Detailed Description of the Invention

The present invention provides a method and system for generating object connections between source objects and sink objects. These connections can be used to support multiple types of event handling mechanisms for objects; the invention provides an underlying connection mechanism architecture for object communication. A source object refers to an object that raises or recognizes an event, and a sink object refers to an object that handles the event. A connection between a source and sink object may be directly initiated by either object or by a third object, referred to as an initiator object. In a typical event handling environment, the source object raises or recognizes an event and notifies the sink object or initiator object by invoking a notification method. If the notification method belongs to the initiator object, then the initiator object is responsible for invoking an appropriate method of the sink object to handle the event.

In a preferred embodiment, the methods and systems of the present invention are implemented on a computer system comprising a central processing unit, memory, and input/output devices. In a preferred embodiment of the present invention, a source object comprises connection point objects and a connection point container object for managing the connection point objects. Preferably, the connection point container object is implemented as part of the source object and the connection point objects are implemented as subobjects of the source object. The subobjects isolate the application independent behavior of the present invention. The connection point container object provides an interface comprising a method that can enumerate the contained connection point objects and a method that can find a connection point object corresponding to a particular interface identifier ("ID"). A connection point object is associated with a certain type of interface (identified by an interface ID) through which it notifies sink objects to which it is connected. A connection point object preferably provides an interface that comprises methods for connecting a notification interface, for disconnecting a previously connected notification interface, and for enumerating the connected notification interfaces. A connection point object preferably can optionally store references to multiple notification interfaces (belonging to one or more sink objects). A connected notification interface acts as an event set. That is, by virtue of the definition of an interface, each object supporting a documented interface must provide a certain set of methods. Thus, when a sink object connects a notification interface, the source object automatically knows what methods are supported by the notification interface. From this perspective, the methods supported loosely correspond to events, and the entire notification interface loosely corresponds to a set of events.

Once connected, the source object can use the connection point objects in a variety of manners. In typical operation, the source object, upon receiving an event notification, consults the connection point object(s) that is (are) associated with the interface ID corresponding to the received event to obtain the

list box object 509. This output port has also been connected to the input port 517 of the code object 504 so that the file list displayed in the multiple selection list box is updated each time the user selects a file. The input port 513 of the button object 510 has been connected to the output port 520 of the code object 505 so that the list of selected files is passed to the button object 510 each time a file is selected. The output port 514 of the button object 510 has been connected to the input port 521 of the code object 506, which contains code that opens each file in the list of selected files once the user has pressed the OK button implemented by button object 510.

Once created using this visual programming environment, the open file dialog box operates by responding to particular system events, for example, events raised from user input devices. For example, when the user selects the open file dialog box 503, a MouseLeftButtonDown selection event is sent to the open file dialog box object 503. Upon receiving this selection event, the open file dialog box object 503 forwards the notification to the code object 504, because the input port 517 of the code object 504 has been previously connected to the output port 516 of the open file dialog box object 503. The code object 504, which implements code for updating the list of displayed files, then sends an updated file list to the multiple selection list box object 509, because the output port 518 of the code object 504 has been previously connected to the input port 511 of the multiple selection list box object 509. Also, when a user selects a file in the list box implemented by the multiple selection list box object 509 using a mouse input device, a MouseLeftButtonDown selection event is sent to the multiple selection list box object 509. This event is then forwarded to the code object 505 to keep track of the user selection because the input port 519 of the code object 505 has been previously connected to the output port 512 of the multiple selection list box object 509. The code object 505 then sends a list of selected files to the button object 510, because the output port 520 of the code object 505 has been previously connected to the input port 513 of the button object 510. In addition, when a user selects the OK button implemented by the button object 510, a system selection event (for example, a MouseLeftButtonDown selection event) is sent to the button object 510. The button object 510 then forwards its output (which in this case is the list of selected files) to the code object 506, because the output port 514 of the button object 510 has been previously connected to the input port 521 of the code object 506. Upon receiving this button selection event, the code object 506 opens the files selected by the user.

In one example application, the present invention can be used to dynamically generate the object connections needed by the visual programming example illustrated in Figure 5. Figure 6 is a block diagram of object connections and data structures after connecting the objects shown in Figure 5 using the present invention. Figure 6 shows four objects: a source object 601, which corresponds to the open file dialog box object 503 in Figure 5 and three sink objects 602-604, which correspond to the code objects 504-506 in Figure 5. The source object 601, corresponding to the open file dialog box object 503, contains subobjects corresponding to the title bar object 508, the multiple selection list box object 509, and the button object 510. (None of the subobjects are shown.) Alternatively, using the present invention, one could create a source object for each of the subobjects contained in the open file dialog box object 503 and then connect each of the source objects with the appropriate code object (sink object).

Because the open file dialog box object 503 deals with system events corresponding to the selection of the open file dialog box object 503, the selection of files within the multiple selection list box object 509, and user selection of the OK button implemented by the button object 510, the source object 601 supports connection point objects associated with different event sets. Specifically, the source object 601 contains a connection point container object 605 and three connection point objects 608, 612, and 615. Connection point object 608 is associated with the IMultipleList interface used to support the multiple selection list box object 509. Connection point object 612 is associated with the IButton interface used to support the button object 510. Connection point object 615 is associated with the IDialog interface used to support the open file dialog box object 503. The connection point container object 605 provides the IConnectionPointContainer interface and maintains a list of pointers to connection point objects. In Figure 6, the list of pointers to connection point objects currently has three elements 606, 607, and 618. Each element contains an indicator of the interface ID associated with the connection point object, a pointer to the IConnectionPoint interface of the connection point object, and a pointer to the next element of the list. One skilled in the art would realize that other data structures could be used to manage the set of created connection point objects. Also, more or less information could be associated with each list element for efficiency reasons. For example, each element need not store the interface ID, as the interface ID is readily accessible from the connection point object.

Each connection point object provides the IConnectionPoint interface and maintains a list of references to notification interfaces belonging to sink objects. A reference to a notification interface of a sink object is added to this list whenever the sink object requests a connection from a connection point object using the

BNSDOCID: <EP 0660231A2>

```
interface IEnumConnections: public IUnknown {
        virtual HRESULT Next (ULONG cConnections, CONNECTDATA *rgpunk,
                              ULONG *lpcFetched) = 0;
        virtual HRESULT Skip (ULONG cConnections) = 0;
        virtual HRESULT Reset ( ) = 0;
        virtual HRESULT Clone (IEnumConnection **ppEnum) = 0;
        }

struct tagCONNECTDATA {
        IUnknown *punk;
        DWORD dwToken;
        } CONNECTDATA;
```

Code Table 3 contains C++ pseudocode for a preferred definition of the interfaces IConnectionPoint and IEnumConnections and the data structure returned by the enumerator interface IEnumConnections. The IConnectionPoint interface contains methods for connecting and disconnecting to the connection point object and for enumerating the notification interfaces connected to the connection point object. The method GetConnectionInterface returns a pointer to the interface ID associated with the connection point object. The method GetConnectionPointContainer returns a pointer to the IConnectionPointContainer interface of the connection point container object containing the connection point object (its parent container object). When the connection point object is instantiated, the creation method of the connection point object is passed a pointer to the connection point container object for future use. The method Advise connects the notification interface specified by the parameter punk to the connection point object and, if successful, returns a unique token identifying the connection in parameter pdwToken. The unique token may be stored persistently. The method Unadvise disconnects the notification interface specified by the input parameter dwToken. The method EnumConnections returns an enumerator interface, an instance of the interface IEnumConnections, for iteration through the connected notification interfaces.

The interface IEnumConnections implements the enumerator used by the IConnectionPoint interface. This enumerator contains a set of methods for enumerating the notification interface connections for a particular connection point object. The two methods of interest include the method Reset, which reinitializes the enumerator to point to the first connected notification interface and the method Next, which returns a pointer to the next connected notification interface. Code Table 3 shows a typical structure definition for the connection information returned by the enumerator method Next referred to as CONNECTDATA.

Code Table 4

```
interface IConnectionPointContainer: public IUnknown {
        virtual HRESULT EnumConnectionPoints (IEnumConnectionPoints **ppEnum) = 0;
        virtual HRESULT FindConnectionPoint (REFIID iid, IConnectionPoint **ppPoint) = 0;
}

interface IEnumConnectionPoints: public IUnknown {
        virtual HRESULT Next (ULONG cConnections, IConnectionPoint *rgpcn,
                              ULONG *lpcFetched) = 0;
        virtual HRESULT Skip (ULONG cConnections) = 0;
        virtual HRESULT Reset ( ) = 0;
        virtual HRESULT Clone (IEnumEmbeddedConnection **ppecn) = 0;
        }
```

Code Table 4 contains C++ pseudocode for preferred definitions of the interfaces IConnectionPointContainer and IEnumConnectionPoints. The IConnectionPointContainer interface implements methods for finding a particular connection point object and for enumerating the set of contained connection point objects. The IEnumConnectionPoints interface implements the enumerator method used by the IConnectionPointContainer interface. The IConnectionPointContainer interface contains a method FindConnectionPoint which returns a pointer to an IConnectionPoint interface given a specified interface ID. The method

the first list element. In step 802, the method GetConnectionInterface of the interface pointed to by the temporary variable is invoked to determine whether the interface ID associated with the connection point object referenced by the temporary variable (the current connection point object) matches the specified interface ID. In step 803, if the returned interface ID matches the specified interface ID, then the method continues at step 804, else continues at step 805. In step 804, the method sets the output parameter to point to the address of the IConnectionPoint interface pointer referenced by the temporary variable, and returns. In step 805, the temporary variable (which points to the current connection point object) is set to point to the IConnectionPoint interface of the next element in the list of instantiated connection point objects. In step 806, if the method has reached the end of the list, then the method continues at step 807, else the method returns to the beginning of the loop in step 801. In step 807, the method determines whether the specified interface ID corresponds to a connection interface that the source object supports, and if so, the method continues at step 808, else returns in error. In step 808, the method instantiates a new connection point object. In step 809, the method inserts the newly instantiated connection point object into the connection point container object's list of connection point objects. In step 810, the method sets the output parameter to point to the address of the newly instantiated connection point object, and returns.

The steps comprising the method FindConnectionPoint in Figure 8 assume that connection point objects are instantiated dynamically as needed. One skilled in the art would recognize that connection point objects can be established dynamically or statically at the discretion of the source object implementation. For example, upon instantiation of the source object, a connection point object corresponding to each connection interface identifier supported by the source object could be instantiated with empty lists of references to notification interfaces. Also, certain steps could be eliminated for efficiency reasons from the method FindConnectionPoint if the connection point container object is implemented with knowledge of the connection point object implementation structure. Such knowledge might typically occur if the source object implementation provides its own implementations for the connection point container object and the connection point objects. In addition, the method FindConnectionPoint assumes that the data structure used to store references to the connection point objects is a list structure as shown in Figure 6. This method could be alternatively written to handle various storage data structures.

Figure 9 is a flow diagram of a method that uses an established connection between a source object and a sink object. Specifically, Figure 9 illustrates a set of steps that could be performed by the source object corresponding to the open file dialog box object 503 in Figure 5 when the source object receives a system selection event indicating that a user has depressed the OK button object 510. This example assumes the connections have been appropriately established as discussed with reference to Figure 6. One skilled in the art would recognize that many other uses of and semantics for the object connection mechanism are possible.

When a user depresses the OK button object 510 in Figure 5, the system sends a selection event to the source object. The source object then invokes some internal routine to respond to the externally raised event. Figure 5 depicts an example of such a routine, which is the method OK_ButtonDown for the IDialogBox interface. The OK_ButtonDown method determines which connection point object corresponds to the interface identifier associated with the raised event and invokes a predetermined method of the notification interfaces connected to the determined connection point object. As described earlier, because the set of events that includes the raised event is represented by an interface, the source object has knowledge of what methods are supported by a connected sink object. Furthermore, in the source object routine handling the raised event (in this case, the OK_ButtonDown method), the source object can determine which particular method of the sink object it prefers to invoke to handle the raised event. In this particular example, the method determines that the method MouseLeftButtonDown of the notification interface corresponding to the interface identifier IID_IButton is preferably invoked to respond to the raised selection event.

In step 901, the method obtains its own IConnectionPointContainer interface using the method QueryInterface. In step 902, the method uses the IConnectionPointContainer interface pointer to invoke the method FindConnectionPoint requesting the connection point object that corresponds to the interface identifier IID_IButton. In step 903, the method invokes the method EnumConnections of the connection point object returned in the previous step to obtain an enumerator for enumerating the contents of the connection point object. In step 904, the method resets the enumerator to start at the beginning of the list of references to notification interfaces. In step 905, the method invokes the method Next of the enumerator to obtain the connection data for the next referenced notification interface. In step 906, if the enumerator indicates no more references to notification interfaces are present, then the method returns, else the method continues in step 907. In step 907, the method calls the method QueryInterface of the IUnknown interface indicated in the connection point data structure requesting the notification interface corresponding

3. The method of claim 1, the connection point interface for connecting to a plurality of sink objects, wherein the steps of receiving a request to connect and storing the indicated notification interface are performed for each sink object, and further including the step of:
invoking the stored notification interface for each sink object.

4. The method of claim 1 wherein each connection point interface connects to a plurality of sink objects, and wherein the step of selecting a connection point interface chooses a connection point interface based upon the notification interface of the sink object.

5. The method of claim 4, the source object having a connection point container object for managing interaction with the plurality of connection point interfaces and wherein the step of selecting a connection point interface includes the substep of requesting a connection point interface from the connection point container object.

6. The method of claim 1, the connection point interface having an advise member function for requesting a connection to the source object, wherein the step of receiving a request to connect is performed under the control of the advise member function of the connection point interface.

7. The method of claim 1 wherein the step of selecting a connection point interface is performed under the control of the source object.

8. The method of claim 1 wherein the step of storing the indicated notification interface is performed under the control of the source object.

9. The method of claim 1, further comprising the step of, under control of the sink object, requesting a connection.

10. The method of claim 9, the computer system having an initiator object for setting up connections between a source object and a sink object, wherein the step of requesting a connection is performed by the initiator object.

11. A method in a computer system for registering a notification interface of a sink object with a source object, the source object having a function member for registering the notification interface of the sink object, the notification interface for communicating with the sink object from the source object, the sink object having a plurality of notification interfaces, the method including the steps of:
selecting a notification interface from the plurality of notification interfaces to register; and
requesting registration of the selected notification interface using the function member of the source object.

12. The method of claim 11, the source object having an advise member function for requesting registration of a notification interface, and wherein the step of requesting registration invokes the advise member function of the source object to make the request.

13. The method of claim 11, the sink object having an IUnknown interface for accessing other interfaces of the sink object, and wherein the step of selecting a notification interface chooses the IUnknown interface of the sink object.

14. The method of claim 9, the computer system having an initiator object for registering a notification interface of a sink object, wherein all steps are performed by the initiator object.

15. A method in a computer system for dynamically generating an object connection between a source object and a sink object, the sink object having a plurality of notification interfaces for communicating with the sink object, the source object having connection point interfaces for connecting the sink object, the method comprising the steps of:
selecting a connection point interface for connecting the source object to the sink object;
selecting a notification interface from the plurality of notification interfaces;
using the selected connection point interface to request that the source object and the sink object be connected, wherein the request indicates the selected notification interface; and

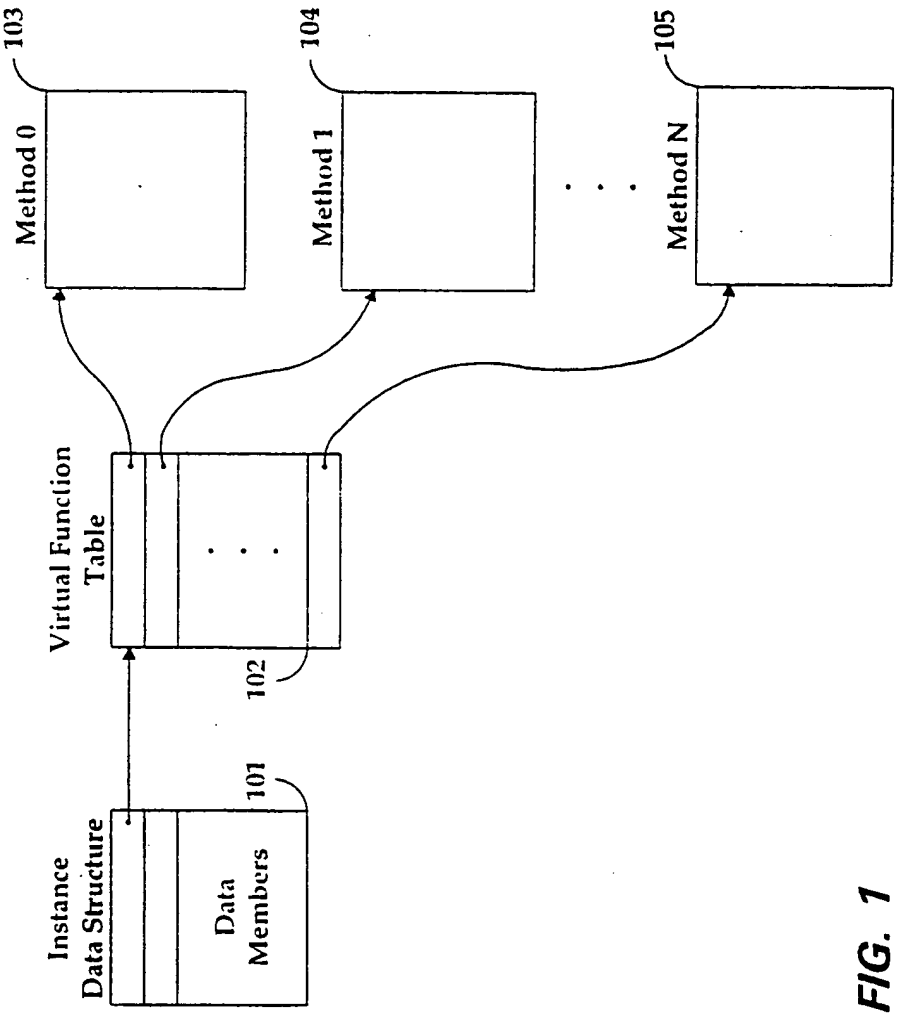
103
Method 0
104
Method 1
105
Method N
Virtual Function Table
102
Instance Data Structure
Data Members
101

FIG. 1

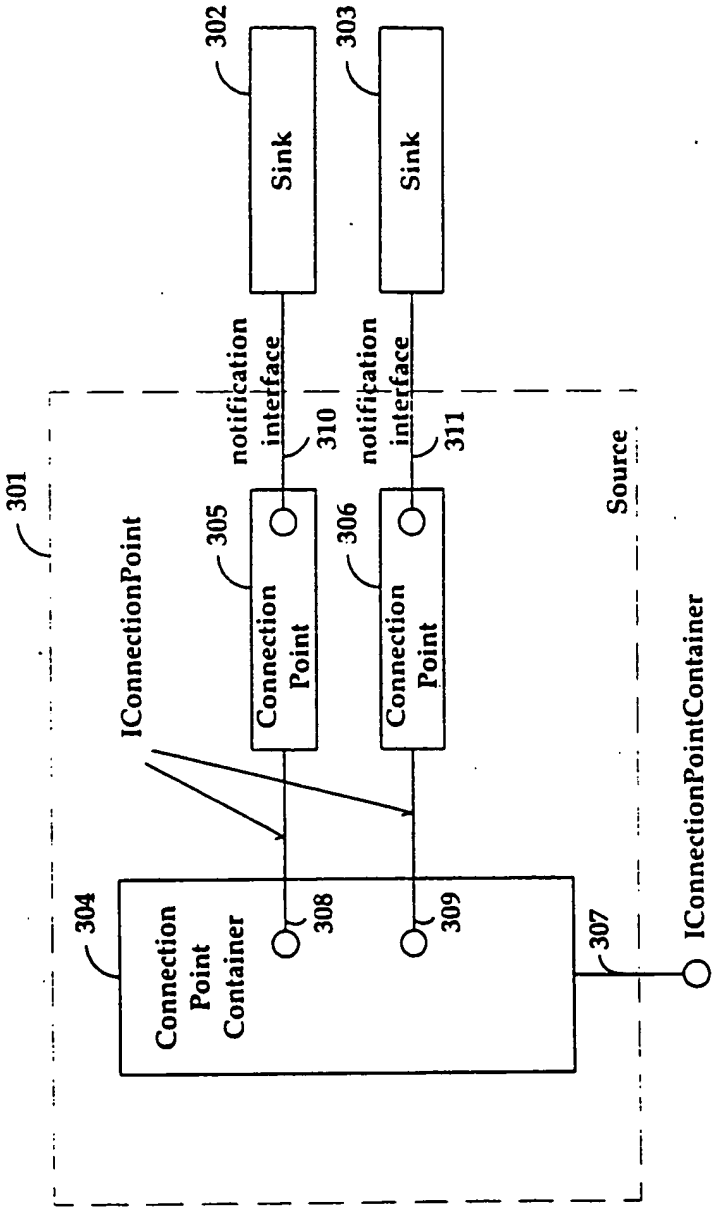

FIG. 3

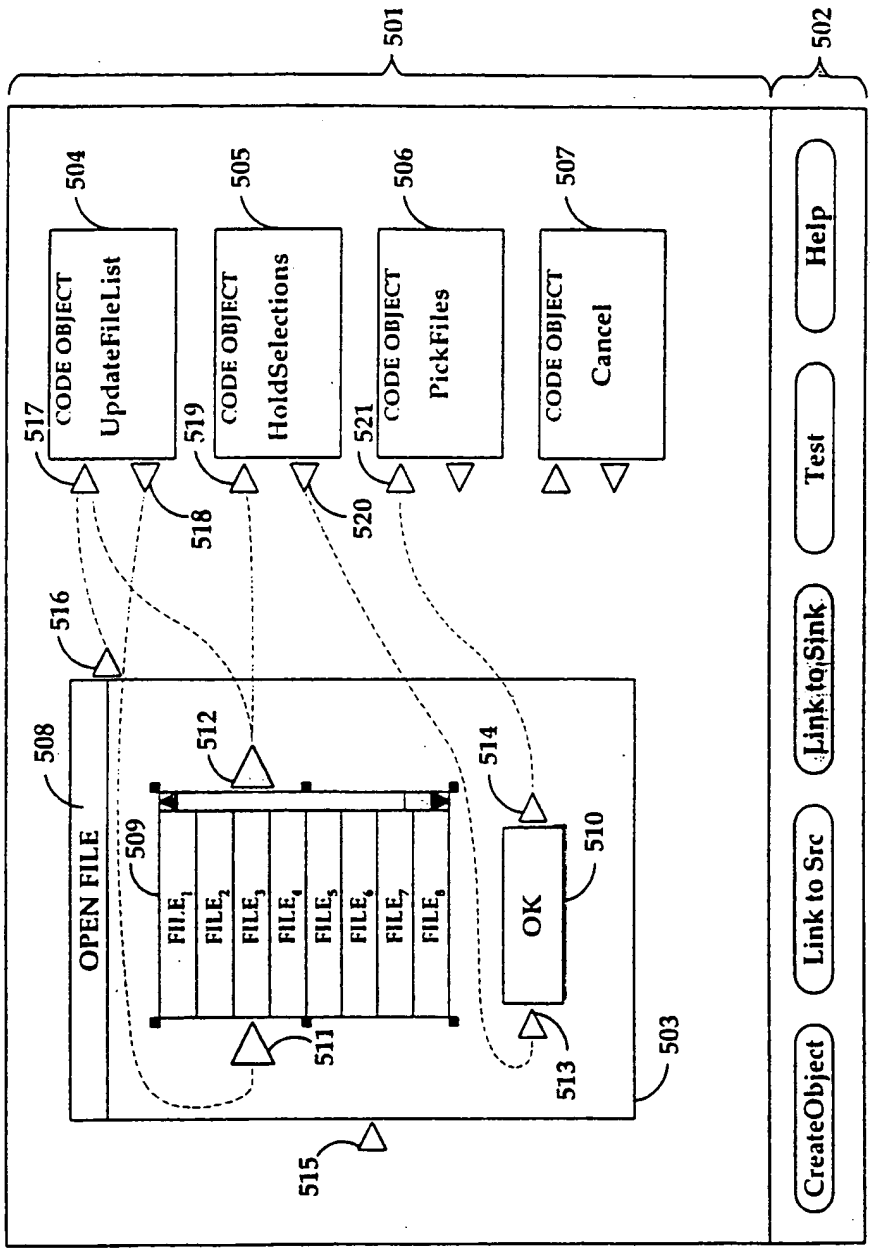
501
502
504
505
506
507
CODE OBJECT
UpdateFileList
CODE OBJECT
HoldSelections
CODE OBJECT
PickFiles
CODE OBJECT
Cancel
517
519
521
518
520
516
508
512
514
509
510
OPEN FILE
FILE1
FILE2
FILE3
FILE4
FILE5
FILE6
FILE7
FILE8
OK
511
513
503
515
Help
Test
Link to Sink
Link to Src
CreateObject

FIG. 5

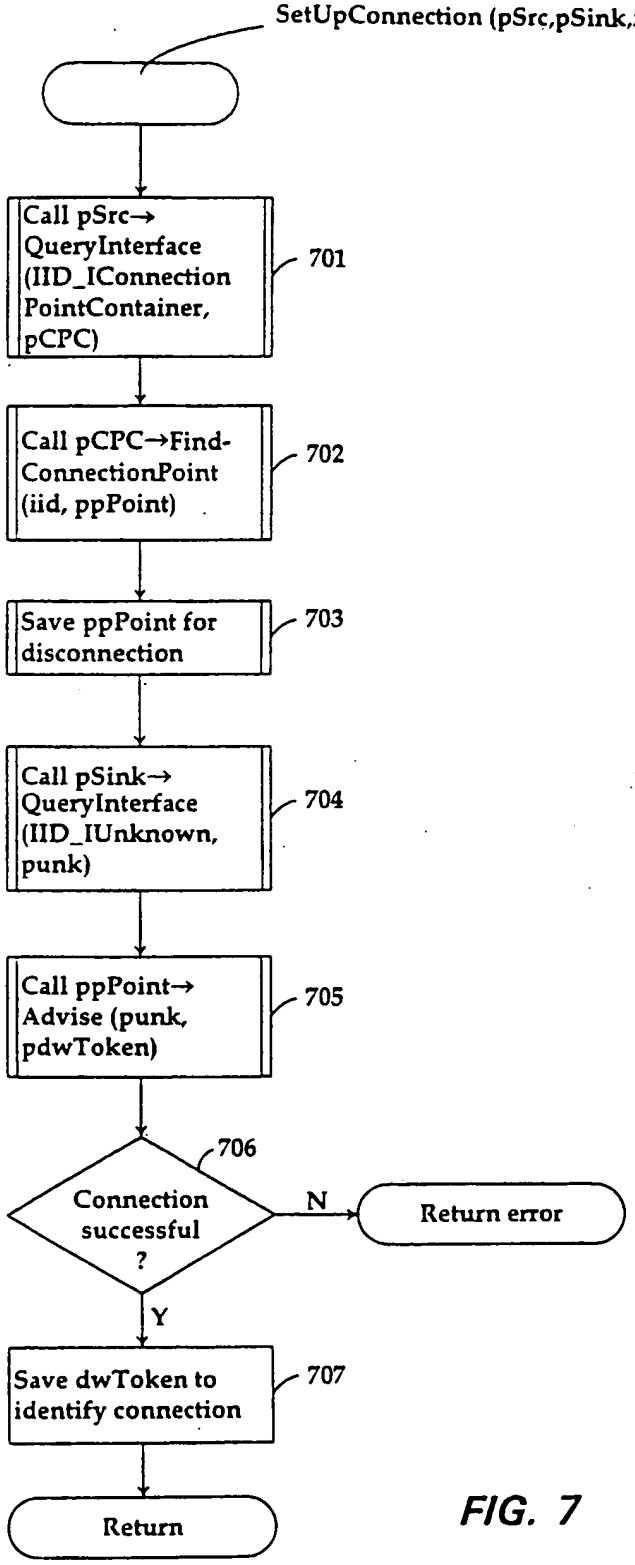

FIG. 7

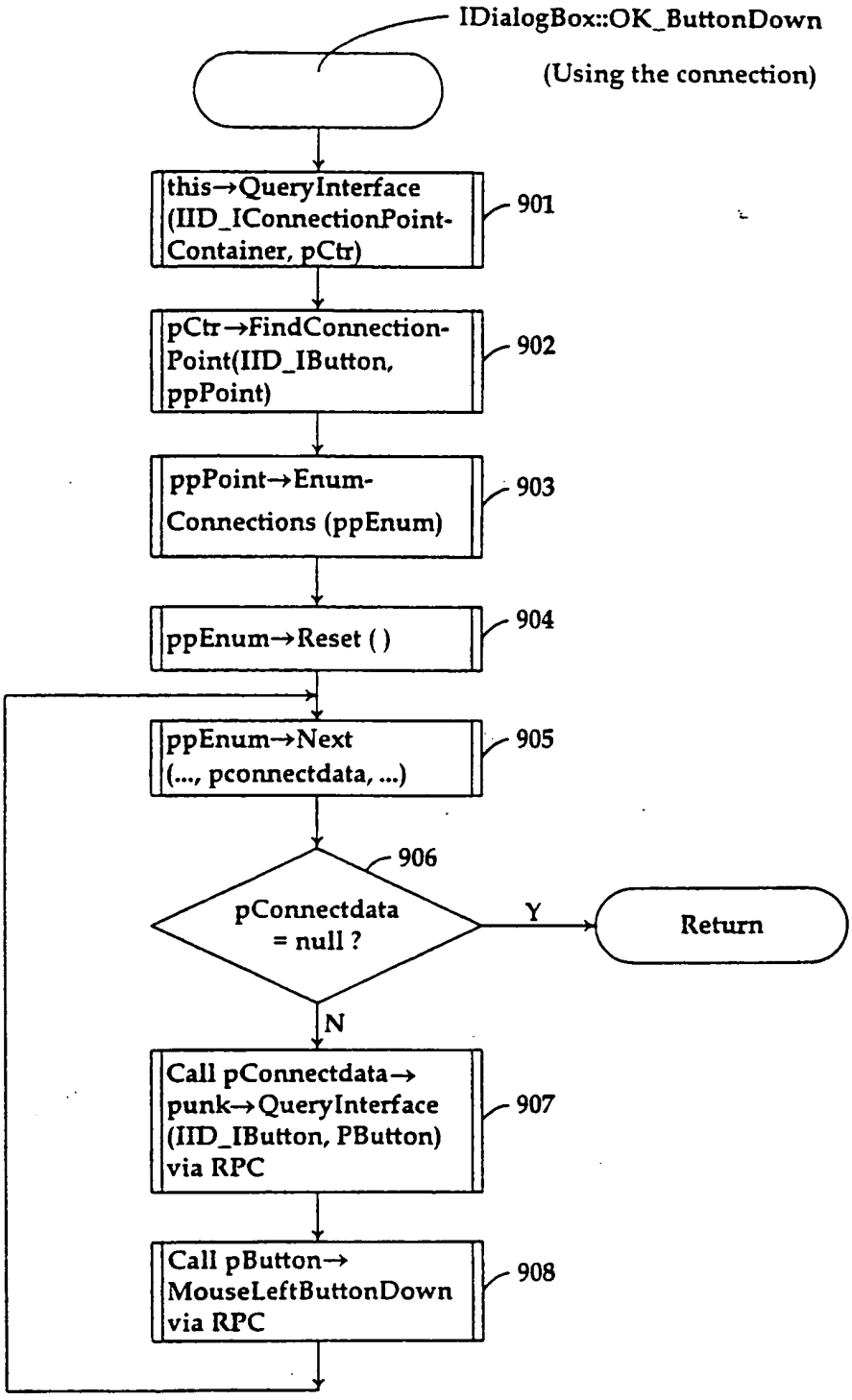

FIG. 9

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 660 231 A3**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **94119549.7**

(51) Int. Cl.[6]: **G06F 9/44**

(22) Date of filing: **09.12.94**

(30) Priority: **13.12.93 US 166976**

(43) Date of publication of application:
28.06.95 Bulletin 95/26

(84) Designated Contracting States:
DE FR GB

(88) Date of deferred publication of the search report:
16.08.95 Bulletin 95/33

(71) Applicant: **MICROSOFT CORPORATION**
One Microsoft Way
Redmond,
Washington 98052-6399 (US)

(72) Inventor: **Stutz, David S.**
19610 N.E. 116th,
Redmond
Washington 98053 (US)
Inventor: **Zimmerman, Christopher A.**
3006 West Lake Sammamish Parkway S.E.
Bellevue,
Washington 98008 (US)

(74) Representative: **Patentanwälte Grünecker,**
Kinkeldey, Stockmair & Partner
Maximilianstrasse 58
D-80538 München (DE)

(54) Method and system for dynamically generating object connections.

(57) A method and system for dynamically generating object connections is provided. In a preferred embodiment, a connection can be generated between a source object and a sink object using a connection point object. A source object has connection point objects where each connection point object corresponds to a particular interface. A sink object implements one or more notification interfaces for connecting to a source object. A connection point object of a source object can connect to multiple notification interfaces, which belong to one or more sink objects. A connection point object keeps track of pointers to the notification interfaces to which it has been connected. In order to generate a connection, a sink object requests from a source object a connection point object corresponding to a particular interface. The source object determines whether it supports such a connection point object, and if so returns a pointer to the connection point interface of the determined connection point object. The sink object then requests to be connected to the connection point object using the returned connection point interface pointer and passes a reference to a notification interface of the sink object corresponding to the particular interface. The connection point object then stores the reference to the notification interface of the sink object, creating a connection between the sink object and the source object. At some later time, the source object can utilize the connection to notify the sink object through the connected notification interfaces.

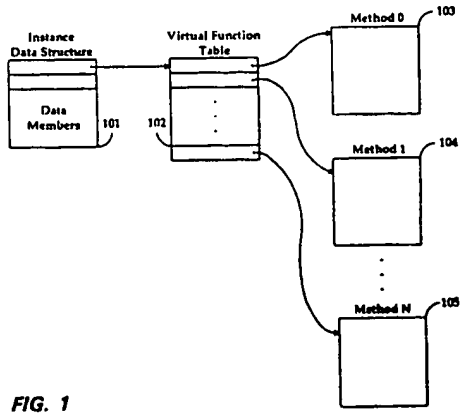

FIG. 1

EP 0 660 231 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 94 11 9549

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
A	PROCEEDINGS CASCON '93, PROCEEDINGS OF CASCON '93, TORONTO, ONT., CANADA, 24-28 OCT. 1993, 1993, OTTAWA, ONT., CANADA, NAT. RES. COUNCIL OF CANADA, CANADA, pages 570-580 vol.1, LAU C 'Using SOM for tool integration' * paragraph 9 *	11	

TECHNICAL FIELDS SEARCHED (Int.Cl.6)

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	26 June 1995	Michel, T

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C04)